REACHING PHYSICAL BOARD ADDRESSES IN A PC FROM WINDOWS PROTECTED MODE

David L. Huffman
Fermilab[1] P.O. Box 500 Batavia, IL 60510

Abstract

This paper will give programming details and examples to show how to reach a physical memory address from within Windows for the purpose of communicating with standard interface cards.

I. INTRODUCTION

Calorimetry electronics development and testing involve collecting and analyzing large amounts of data. The system (Fig. 1) is required to control various conditions depending on the type of test. The D0 Calorimeter electronics test system, dubbed 'The 5000 Channel Test,' must monitor and control 48x24x4=4608, ~5000, preamp channels, five system power supplies with about 12 voltages and currents each, and other digital and analog information. This information, all of which is located in a VME crate, is read hundreds of times in order to get statistical information. Code that could process this large data set in real time would be very desirable.

All information is made available in a single VME crate. A bus-to-bus interface card[2] maps VME memory to PC memory, allowing the PC to perform all the testing.

II. OVERVIEW

	Linear	Address	
Kilobytes	Decimal	Hex	Description
0	0	00000000	
64	65536	00010000	
128	131072	00020000	
192	196608	00030000	
256	262144	00040000	
320	327680	00050000	
384	393216	00060000	
448	458752	00070000	
512	524288	00080000	
576	589824	00090000	
640	655360	000A0000	VIDEO
704	720896	000B0000	VIDEO
768	786432	000C0000	VIDEO
832	851968	000D0000	INTERFACE CARD
896	917504	000E0000	
960	983040	000F0000	BASIC & BOOT BIOS
1024	1048576	00100000	BASIC & BOOT BIOS

TABLE 1.
A typical first megabyte of memory in a PC.

The typical arrangement, in the past, was to 'fit' the interface card into open memory somewhere above the video adapter as seen in table 1. For this arrangement the software would simply peek or poke the location to perform the necessary operation.

This arrangement has some major drawbacks. The open memory locations become hard to find as other cards are added to the system. Usually the network cards like to live in these open locations and most of our systems are networked together.

Another limitation is that real-mode applications require each code and data segment to be less than 64K, and it must run in the first megabyte of memory. The familiar SEGMENT/OFFSET addressing, which when combined makes up a 20 bit address, does not allow addressing memory beyond the first megabyte since 2^{20}=1048576. Huge arrays in memory are not possible in real mode. We need to process large blocks of data. With real-mode coding disk files are manipulated to calculate statistical information on large data blocks. This makes the applications run slowly due to disk access times.

There is no multitasking since only one application can run at a time.

III. MOVING UP IN MEMORY

If we could move the interface card above the normal RAM of the system we would make precious memory available. If we use Windows as a protected-mode DOS extender we get large chunks of memory to program with and multitasking capability.

The method chosen to accomplish memory access involves using the DPMI[3] function 800. This function allows any physical location to be accessed, thus allowing communication with cards located anywhere in memory space.

The interface card is set to address 00C00000h (12 meg), above the normal 8 megabyte of RAM typically available on our systems. Table 2 shows this placement in a typical system.

The trick now is to gain access to the physical memory of the card at its new location.

IV. HOW TO GET TO PHYSICAL MEMORY

[1] Work supported by the U.S. Department of Energy under contract No. DE-AC02-76CHO3000

[2] We are using a Bit3 Computer Corporation type 403 or 406 VMEbus Adaptor card.

[3] DOS PROTECTED MODE INTERFACE (see ref [1]).

The Windows environment uses the Intel[4] 386 and higher, processor in virtual mode and provides access to all available RAM memory. This memory is, however, virtual and physical addressing is not allowed.

The problem with programming in this environment is that you are not allowed to peek[5] (read) or poke (write) any physical locations since they are made virtual by the processors' protection feature. A General Protection Fault (exception 13) is the result of such an effort. In order to read and write to physical memory location we need to use the DPMI services.

	Linear	Address	
Megabyte	Decimal	Hex	Description
1	1048576	00100000	DOS/BIOS/VIDEO
2	2097152	00200000	RAM
3	3145728	00300000	RAM
4	4194304	00400000	RAM
5	5242880	00500000	RAM
6	6291456	00600000	RAM
7	7340032	00700000	RAM
8	8388608	00800000	RAM
9	9437184	00900000	-
10	10485760	00A00000	-
11	11534336	00B00000	-
12	12582912	00C00000	INTERFACE CARD
13	13631488	00D00000	-
14	14680064	00E00000	-
15	15728640	00F00000	-
16	16777216	01000000	-

TABLE 2.
Machine with 8 megabytes of RAM.

The following code excerpt, called **MagicCode()**, shows how the DPMI function 0800h is used to provide access to any memory location.

```
unsigned char far *pMem;
static UINT MagicCode( void )
{
UINT  __DS;
DWORD dwLinearAddress;
UINT ilow, ihi;

 if(PhysicalAddress >= 0x100000) {
     ilow = LOWORD( PhysicalAddress );
     ihi = HIWORD( PhysicalAddress );
     _asm {
         mov ax, 0800h
         mov bx, ihi
         mov cx, ilow
         mov si, 0003h
         mov di, 0000h
         int 31h
         mov ihi, bx
         mov ilow,  cx
     }
     dwLinearAddress = MAKELONG( ilow, ihi);
}
else{
     dwLinearAddress = PhysicalAddress;
}
_asm mov __DS, ds;
Sel = AllocSelector( __DS );
SetSelectorBase( Sel, dwLinearAddress );
SetSelectorLimit( Sel, 0xFFFF );
FP_SEG(pMem) = sel;
FP_OFF(pMem)= iOffset;
FreeSelector(sel);
}
```

The **PhysicalAddress** variable is set to the 32 bit address of the desired memory location. It is separated into a high and low word that is loaded into the **bx** and **cx** registers. The **ax** register is loaded with the function call number 0800h. The **si** and **dx** registers are set to the size of the region that will be mapped in bytes. After an interrupt 31h is processed the **bx:cx** registers will contain the linear address that is mapped to the physical address requested. Now that the linear address is known we can use it to get a pointer to the desired memory. Using Windows function **AllocSelector()** a new selector is copied using an existing selector. The base of the selector is set using the returned linear address **dwLinearAddress**. Note that the linear address is equal to the physical address when address is less than 1 megabyte and in-line assembly code for the DPMI function is not used for those addresses.

With the selector copied a pointer can be formed using **FP_SEG( )** and **FP_OFF( )** functions to get at the memory requested.

An application such as a memory dump, could loop around the FP_OFF( ) function to read/write several different locations in the same segment before releasing the selector as seen below.

```
...
FP_SEG(pRealMem) = sel;
 for(i=0 ; i < iCol*2  ; i += 2)
 {
     FP_OFF(pRealMem) = i;
     iodata = *pRealMem;
// read memory location!!!!!!!!
     if(bytebutton)
     {   //display as byte info
     wsprintf(achTemp, "%02X %02X\t",
         LOBYTE( iodata ), HIBYTE( iodata ));
     }
     else    // else display as WORD info
     wsprintf(achTemp, "%04X\t", iodata );
     lstrcat(achString[linecnt], achTemp);
// ***print the ASCI equivalent of the data  ***
     wsprintf(achTemp, "%c%c",
         LOBYTE( iodata ), HIBYTE ( iodata ));
 lstrcat(achChar[linecnt], achTemp);
 }
 lstrcat(achString[linecnt], achChar[linecnt]);
// put the two strings together
 FreeSelector(sel);
...
```

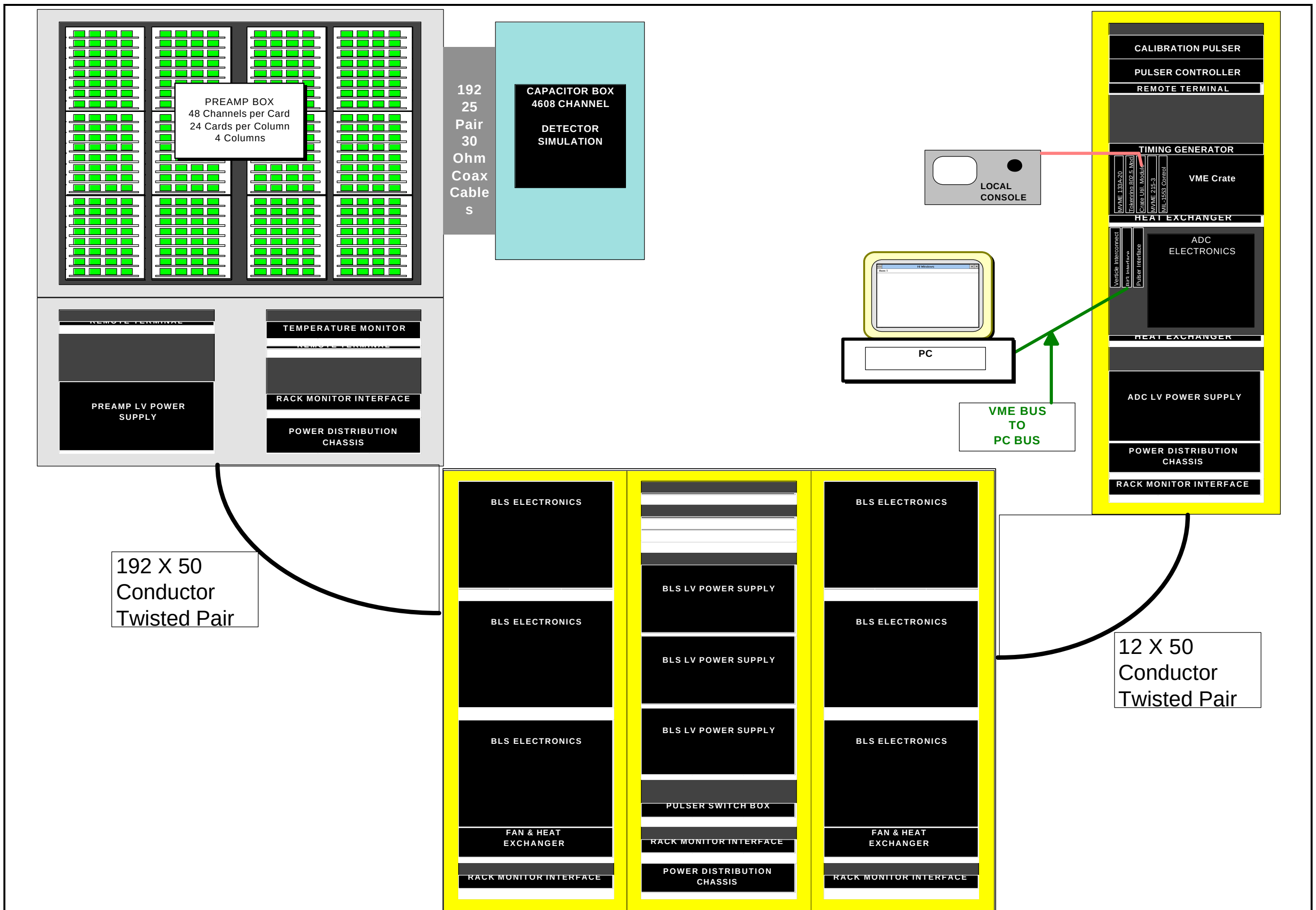
This may seem like a lot of code, but it is necessary to circumvent, in a meaningful way, what the processor protection is preventing.

V. CONCLUSIONS

The DPMI function 800h is not the only method available for reaching physical addresses. It does, however, provided the bridge we needed to make the card addressable from Windows. It has also allowed advantages previously unavailable from DOS code previously used.

[4] A registered trade mark of the Intel Corporation, processor type 386 and higher.

[5] These functions do in fact work from Windows for values below 1 megabyte.



The 5000 Channel Test system.
Figure 1.

First it lets us place the interface card at any memory space accessible to the hardware. This would be within the first 16 megabytes of address space. This relieves the congestion in the first megabyte of memory.

Second it moves software development into the Windows environment, which benefits the end user and allows the PC to run several applications together.

The **MagicCode()** has been used by other programmers to jump start their applications as they move into the Windows environment. This basic building block was used in programs that monitor and set values in the D0 Calorimeter 5000 channel test system.

Recently code was written which reads or writes VME memory directly into or from an Excel[6] worksheet cell. This code, written as a DLL[7], allows users the flexibility of viewing and manipulating information any way they wish.

Looking to the future, there is a way to reach the same VME memory through the use of SOCKETS[8]. Although slower than the direct link of an interface card it has the advantage of being accessible from any ethernet node.

[6] Excel is a registered trademark of Microsoft Corp.

[7] Dynamic Link Libraries are like DOS TSRs and are available to all Windows Applications.

[8] Sockets will be a standard part of 'Windows 95' the next Windows generation. It is currently available for Windows for Workgroups 3.11.

VI. References

[1] The DPMI Committee, *DOS PROTECTED MODE INTERFACE (DPMI) SPECIFICATION*, Version 0.9, Software Focus Group, Intel Corporation, NW1-18, May 15, 1990.

[2] Programming Windows, *WINDOWS MAGAZINE*, July 1992, pp. 272.

[3] Andrew Schulman, *PC MAGAZINE*, Lab Notes "The Programming Challenge of Windows Protected Mode", June 25, 1991, pp. 371-389.

[4] Kerry Loynd, *DR. DOBB'S JOURNAL*, Mixing Real- and Protected-Mode Code, February 1992.

[5] Paul Bonneau, WINDOWS/DOS DEVELOPER's JOURNAL, Memory Management, Far Pointers for Huge Memory, September 1994, Vol. 5, No.9, pp. 6-22.